# Dependently Typed Programming in Agda

Ulf Norell[1] and James Chapman[2] *

[1] Chalmers University, Gothenburg
ulfn@chalmers.se
[2] Institute of Cybernetics, Tallinn
james@cs.ioc.ee

## 1 Introduction

In Hindley-Milner style languages, such as Haskell and ML, there is a clear separation between types and values. In a dependently typed language the line is more blurry – types can contain (*depend on*) arbitrary values and appear as arguments and results of ordinary functions.

The standard example of a dependent type is the type of lists of a given length: `Vec A n`. Here `A` is the type of the elements and `n` is the length of the list. Many languages allow you to define lists (or arrays) of a given size, but what makes `Vec` a true dependent type is that the length of the list can be an arbitrary term, which need not be known at compile time.

Since dependent types allows types to talk about values, we can encode properties of values as types whose elements are proofs that the property is true. This means that a dependently typed programming language can be used as a logic. In order for this logic to be consistent we need to require programs to be total, i.e. they are not allowed to crash or non-terminate.

The rest of these notes are structured as follows: Section 2 introduces the dependently typed language Agda and its basic features, and Section 3 explains a couple of programming techniques made possible by the introduction of dependent types.

## 2 Agda Basics

Agda is a dependently typed language based on intuitionistic type theory[4]. Its current version (Agda 2) is a complete rewrite instigated by Ulf Norell during his PhD[6] at Chalmers University in Gothenburg. This section introduces the basic features of Agda and how they can be employed in the construction of dependently typed programs. Information on how

* Please send bug reports for this tutorial to James.

to obtain the Agda system and further details on the topics discussed here can be found on the Agda wiki [2].

This section is a literate Agda file which can be compiled by the Agda system. Hence, we need to start at the beginning: Every Agda file contains a single top-level module whose name corresponds to the name of the file. In this case the file is called `AgdaBasics.lagda`[3].

```
module AgdaBasics where
```

The rest of your program goes inside the top-level module. Let us start by defining some simple datatypes and functions.

### 2.1 Datatypes and pattern matching

Similar to languages like Haskell and ML, a key concept in Agda is pattern matching over algebraic datatypes. With the introduction of dependent types pattern matching becomes even more powerful as we shall see in Section 2.4 and Section 3. But for now, let us start with simply typed functions and datatypes.

Datatypes are introduced by a `data` declaration, giving the name and type of the datatype as well as the constructors and their types. For instance, here is the type of booleans

```
data Bool : Set where
  true  : Bool
  false : Bool
```

The type of `Bool` is `Set`, the type of small[4] types. Functions over `Bool` can be defined by pattern matching in a way familiar to Haskell programmers:

```
not : Bool -> Bool
not true  = false
not false = true
```

Agda functions are not allowed to crash, so a function definition must cover all possible cases. This will be checked by the type checker and an error is raised if there are missing cases.

[3] Literate Agda files have the extension `lagda` and ordinary Agda files have the extension `agda`.

[4] There is hierarchy of increasingly large types. The type of `Set` is `Set1`, whose type is `Set2`, and so on.

In Haskell and ML the type of `not` can be inferred from the defining clauses and so in these languages the type signature is not required. However, in the presence of dependent types this is no longer the case and we are forced to write down the type signature of `not`. This is not a bad thing, since by writing down the type signature we allow the type checker, not only to tell us when we make mistakes, but also to guide us in the construction of the program. When types grow more and more precise the dialog between the programmer and the type checker gets more and more interesting.

Another useful datatype is the type of (unary) natural numbers.

```
data Nat : Set where
  zero : Nat
  suc  : Nat -> Nat
```

Addition on natural numbers can be defined as a recursive function.

```
_+_ : Nat -> Nat -> Nat
zero  + m = m
suc n + m = suc (n + m)
```

In the same way as functions are not allowed to crash, they must also be terminating. To guarantee termination recursive calls have to be made on structurally smaller arguments. In this case `_+_` passes the termination checker since the first argument is getting smaller in the recursive call (`n` $<$ `suc n`). Let us define multiplication while we are at it

```
_*_ : Nat -> Nat -> Nat
zero  * m = zero
suc n * m = m + n * m
```

Agda supports a flexible mechanism for mixfix operators. If a name of a function contains underscores (`_`) it can be used as an operator with the arguments going where the underscores are. Consequently, the function `_+_` can be used as an infix operator writing `n + m` for `_+_ n m`. There are (almost) no restrictions on what symbols are allowed as operator names, for instance we can define

```
_or_ : Bool -> Bool -> Bool
false or x = x
true  or _ = true

if_then_else_ : {A : Set} -> Bool -> A -> A -> A
if true  then x else y = x
if false then x else y = y
```

In the second clause of the `_or_` function the underscore is a wildcard pattern, indicating that we don't care what the second argument is and we can't be bothered giving it a name. This, of course, means that we cannot refer to it on the right hand side. The precedence and fixity of an operator can be declared with an `infix` declaration:

```
infixl 60 _*_
infixl 40 _+_
infixr 20 _or_
infix  5 if_then_else_
```

There are some new and interesting bits in the type of `if_then_else_`. For now, it is sufficient to think about `{A : Set} ->` as declaring a polymorphic function over a type `A`. More on this in Sections 2.2 and 2.3.

Just as in Haskell and ML datatypes can be parameterised by other types. The type of lists of elements of an arbitrary type is defined by

```
infixr 40 _::_
data List (A : Set) : Set where
  []   : List A
  _::_ : A -> List A -> List A
```

Again, note that Agda is quite liberal about what is a valid name. Both `[]` and `_::_` are accepted as sensible names. In fact, Agda names can contain arbitrary non-whitespace unicode characters, with a few exceptions, such as parenthesis and curly braces. So, if we really wanted (which we don't) we could define the list type as

```
data _⋆ (α : Set) : Set where
  ε : α ⋆
  _◁_ : α -> α ⋆ -> α ⋆
```

This liberal policy of names means that being generous with whitespace becomes important. For instance, `not:Bool->Bool` would not be a valid type signature for the `not` function, since it is in fact a valid name.

### 2.2 Dependent functions

Let us now turn our attention to dependent types. The most basic dependent type is the dependent function type, where the result type depends on the value of the argument. In Agda we write `(x : A) -> B` for the type of functions taking an argument `x` of type `A` and returning a result of type `B`, where `x` may appear in `B`. A special case is when `x` itself is a type. For instance, we can define

```
identity : (A : Set) -> A -> A
identity A x = x

zero' : Nat
zero' = identity Nat zero
```

This is a dependent function taking a type argument `A` and an element of `A` and returns the element. This is how polymorphic functions are encoded in Agda. Here is an example of a more intricate dependent function; the function which takes a dependent function and applies it to an argument:

```
apply : (A : Set)(B : A -> Set) ->
        ((x : A) -> B x) -> (a : A) -> B a
apply A B f a = f a
```

Agda accepts some short hands for dependent function types:

- `(x : A)(y : B) -> C` for `(x : A) -> (y : B) -> C`, and
- `(x y : A) -> B` for `(x : A)(y : A) -> B`.

The elements of dependent function types are lambda terms which may carry explicit type information. Some alternative ways to define the identity function above are:

```
identity₂ : (A : Set) -> A -> A
identity₂ = \A x -> x

identity₃ : (A : Set) -> A -> A
identity₃ = \(A : Set)(x : A) -> x

identity₄ : (A : Set) -> A -> A
identity₄ = \(A : Set) x -> x
```

### 2.3 Implicit arguments

We saw in the previous section how dependent functions taking types as arguments could be used to model polymorphic types. The thing with polymorphic functions, however, is that you don't have to say at which type you want to apply it – that is inferred by the type checker. However, in the example of the identity function we had to explicitly provide the type argument when applying the function. In Agda this problem is solved by a general mechanism for *implicit arguments*. To declare a function argument implicit we use curly braces instead of parenthesis in the type: `{x : A} -> B` means the same thing as `(x : A) -> B` except that when

you use a function of this type the type checker will try to figure out the argument for you.

Using this syntax we can define a new version of the identity function, where you don't have to supply the type argument.

```
id : {A : Set} -> A -> A
id x = x

true' : Bool
true' = id true
```

Note that the type argument is implicit both when the function is applied and when it is defined.

There are no restrictions on what arguments can be made implicit, nor are there any guarantees that an implicit argument can be inferred by the type checker. For instance, we could be silly and make the second argument of the identity function implicit as well:

```
silly : {A : Set}{x : A} -> A
silly {_}{x} = x

false' : Bool
false' = silly {x = false}
```

Clearly, there is no way the type checker could figure out what the second argument to `silly` should be. To provide an implicit argument explicitly you use the implicit application syntax `f {v}`, which gives `v` as the left-most implicit argument to `f`, or as shown in the example above, `f {x = v}`, which gives `v` as the implicit argument called `x`. The name of an implicit argument is obtained from the type declaration.

Conversely, if you want the type checker to fill in a term which needs to be given explicitly you can replace it by an underscore. For instance,

```
one : Nat
one = identity _ (suc zero)
```

It is important to note that the type checker will not do any kind of search in order to fill in implicit arguments. It will only look at the typing constraints and perform unification[5].

Even so, a lot can be inferred automatically. For instance, we can define the fully dependent function composition. (Warning: the following type is not for the faint of heart!)

[5] Miller pattern unification to be precise.

```
_∘_ : {A : Set}{B : A -> Set}{C : (x : A) -> B x -> Set}
      (f : {x : A}(y : B x) -> C x y)(g : (x : A) -> B x)
      (x : A) -> C x (g x)
(f ∘ g) x = f (g x)

plus-two = suc ∘ suc
```

The type checker can figure out the type arguments `A`, `B`, and `C`, when we use `_∘_`.

We have seen how to define simply typed datatypes and functions, and how to use dependent types and implicit arguments to represent polymorphic functions. Let us conclude this part by defining some familiar functions.

```
map : {A B : Set} -> (A -> B) -> List A -> List B
map f []        = []
map f (x :: xs) = f x :: map f xs

_++_ : {A : Set} -> List A -> List A -> List A
[]        ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

### 2.4 Datatype families

So far, the only use we have seen of dependent types is to represent polymorphism, so let us look at some more interesting examples. The type of lists of a certain length, mentioned in the introduction, can be defined as follows:

```
data Vec (A : Set) : Nat -> Set where
  []   : Vec A zero
  _::_ : {n : Nat} -> A -> Vec A n -> Vec A (suc n)
```

This declaration introduces a number of interesting things. First, note that the type of `Vec A` is `Nat -> Set`. This means that `Vec A` is a family of types indexed by natural numbers. So, for each natural number `n`, `Vec A n` is a type. The constructors are free to construct elements in an arbitrary type of the family. In particular, `[]` constructs an element in `Vec A zero` and `_::_` an element in `Vec A (suc n)` for some `n`.

There is a distinction between *parameters* and *indices* of a datatype. We say that `Vec` is parameterised by a type `A` and indexed over natural numbers.

In the type of `_::_` we see an example of a dependent function type. The first argument to `_::_` is an implicit natural number `n` which is the

length of the tail. We can safely make `n` implicit since the type checker can infer it from the type of the third argument.

Finally, note that we chose the same constructor names for `Vec` as for `List`. Constructor names are not required to be distinct between different datatypes.

Now, the interesting part comes when we start pattern matching on elements of datatype families. Suppose, for instance, that we want to take the head of a non-empty list. With the `Vec` type we can actually express the type of non-empty lists, so we define `head` as follows:

```
head : {A : Set}{n : Nat} -> Vec A (suc n) -> A
head (x :: xs) = x
```

This definition is accepted by the type checker as being exhaustive, despite the fact that we didn't give a case for `[]`. This is fortunate, since the `[]` case would not even be type correct – the only possible way to build an element of `Vec A (suc n)` is using the `_::_` constructor.

The rule for when you have to include a particular case is very simple: *if it is type correct you have to include it.*

**Dot patterns** Here is another function on `Vec`:

```
vmap : {A B : Set}{n : Nat} -> (A -> B) -> Vec A n -> Vec B n
vmap f []       = []
vmap f (x :: xs) = f x :: vmap f xs
```

Perhaps surprisingly, the definition map on `Vec` is exactly the same as on `List`, the only thing that changed is the type. However, something interesting is going on behind the scenes. For instance, what happens with the length argument when we pattern match on the list? To see this, let us define new versions of `Vec` and `vmap` with fewer implicit arguments:

```
data Vec₂ (A : Set) : Nat -> Set where
  nil  : Vec₂ A zero
  cons : (n : Nat) -> A -> Vec₂ A n -> Vec₂ A (suc n)

vmap₂ : {A B : Set}(n : Nat) -> (A -> B) -> Vec₂ A n -> Vec₂ B n
vmap₂ .zero    f nil           = nil
vmap₂ .(suc n) f (cons n x xs) = cons n (f x) (vmap₂ n f xs)
```

What happens when we pattern match on the list argument is that we learn things about its length: if the list turns out to be `nil` then the length argument must be `zero`, and if the list is `cons n x xs` then the only type correct value for the length argument is `suc n`. To indicate that

the value of an argument has been deduced by type checking, rather than observed by pattern matching it is prefixed by a dot (`.`).

In this example we could choose to define `vmap` by first pattern matching on the length rather than on the list. In that case we would put the dot on the length argument of `cons`[6]:

```
vmap₃ : {A B : Set}(n : Nat) -> (A -> B) -> Vec₂ A n -> Vec₂ B n
vmap₃ zero    f nil            = nil
vmap₃ (suc n) f (cons .n x xs) = cons n (f x) (vmap₃ n f xs)
```

The rule for when an argument should be dotted is: *if there is a unique type correct value for the argument it should be dotted.*

In the example above, the terms under the dots were valid patterns, but in general they can be arbitrary terms. For instance, we can define the image of a function as follows:

```
data Image_∋_ {A B : Set}(f : A -> B) : B -> Set where
  im : (x : A) -> Image f ∋ f x
```

Here we state that the only way to construct an element in the image of `f` is to pick an argument `x` and apply `f` to `x`. Now if we know that a particular `y` is in the image of `f` we can compute the inverse of `f` on `y`:

```
inv : {A B : Set}(f : A -> B)(y : B) -> Image f ∋ y -> A
inv f .(f x) (im x) = x
```

**Absurd patterns** Let us define another datatype family, namely the family of numbers smaller than a given natural number.

```
data Fin : Nat -> Set where
  fzero : {n : Nat} -> Fin (suc n)
  fsuc  : {n : Nat} -> Fin n -> Fin (suc n)
```

Here `fzero` is smaller than `suc n` for any `n` and if `i` is smaller than `n` then `fsuc i` is smaller than `suc n`. Note that there is no way of constructing a number smaller than zero. When there are no possible constructor patterns for a given argument you can pattern match on it with the absurd pattern `()`:

```
magic : {A : Set} -> Fin zero -> A
magic ()
```

[6] In fact the dot can be placed on any of the `n`s. What is important is that there is a unique binding site for each variable in the pattern.

Using an absurd pattern means that you do not have to give a right hand side, since there is no way anyone could provide an argument to your function. One might think that the clause would not have to be given at all, that the type checker would see that the matching is exhaustive without any clauses, but remember that a case can only be omitted if there is no type correct way of writing it. In the case of `magic` a perfectly type correct left hand side is `magic x`.

It is important to note that an absurd pattern can only be used if there are no valid constructor patterns for the argument, it is not enough that there are no closed inhabitants of the type[7]. For instance, if we define

```
data Empty : Set where
  empty : Fin zero -> Empty
```

Arguments of type `Empty` can not be matched with an absurd pattern, since there is a perfectly valid constructor pattern that would do: `empty x`. Hence, to define the `magic` function for `Empty` we have to write

```
magic' : {A : Set} -> Empty -> A
magic' (empty ())
-- magic' ()  -- not accepted
```

Now, let us define some more interesting functions. Given a list of length `n` and a number `i` smaller than `n` we can compute the `i`th element of the list (starting from 0):

```
_!_ : {n : Nat}{A : Set} -> Vec A n -> Fin n -> A
[]        ! ()
(x :: xs) ! fzero    = x
(x :: xs) ! (fsuc i) = xs ! i
```

The types ensure that there is no danger of indexing outside the list. This is reflected in the case of the empty list where there are no possible values for the index.

The `_!_` function turns a list into a function from indices to elements. We can also go the other way, constructing a list given a function from indices to elements:

```
tabulate : {n : Nat}{A : Set} -> (Fin n -> A) -> Vec A n
tabulate {zero}  f = []
tabulate {suc n} f = f fzero :: tabulate (f ∘ fsuc)
```

Note that `tabulate` is defined by recursion over the length of the result list, even though it is an implicit argument. There is in general no correspondance between implicit data and computationally irrelevant data.

[7] Since checking type inhabitation is undecidable.

## 2.5 Programs as proofs

As mentioned in the introduction, Agda's type system is sufficiently powerful to represent (almost) arbitrary propositions as types whose elements are proofs of the proposition. Here are two very simple propositions, the true proposition and the false proposition:

```
data   False : Set where
record True  : Set where

trivial : True
trivial = _
```

The false proposition is represented by the datatype with no constructors and the true proposition by the record type with no fields (see Section 2.8 for more information on records). The record type with no fields has a single element which is the empty record. We could have defined `True` as a datatype with a single element, but the nice thing with the record definition is that the type checker knows that there is a unique element of `True` and will fill in any implicit arguments of type `True` with this element. This is exploited in the definition of `trivial` where the right hand side is just underscore. If you nevertheless want to write the element of `True`, the syntax is `record{}`.

These two propositions are enough to work with decidable propositions. We can model decidable propositions as booleans and define

```
isTrue : Bool -> Set
isTrue true  = True
isTrue false = False
```

Now, `isTrue b` is the type of proofs that `b` equals `true`. Using this technique we can define the safe list lookup function in a different way, working on simply typed lists and numbers.

```
_<_ : Nat -> Nat -> Bool
_     < zero  = false
zero  < suc n = true
suc m < suc n = m < n

length : {A : Set} -> List A -> Nat
length []       = zero
length (x :: xs) = suc (length xs)

lookup : {A : Set}(xs : List A)(n : Nat) ->
         isTrue (n < length xs) -> A
lookup []       n       ()
lookup (x :: xs) zero    p = x
lookup (x :: xs) (suc n) p = lookup xs n p
```

In this case, rather than there being no index into the empty list, there is no proof that a number `n` is smaller than `zero`. In this example using indexed types to capture the precondition is a little bit nicer, since we don't have to pass around an explicit proof object, but some properties cannot be easily captured by indexed types, in which case this is a nice alternative.

We can also use datatype families to define propositions. Here is a definition of the identity relation

```
data _==_ {A : Set}(x : A) : A -> Set where
  refl : x == x
```

For a type `A` and an element `x` of `A`, we define the the family of proofs of "being equal to `x`". This family is only inhabited at index `x` where the single proof is `refl`.

Another example is the less than or equals relation on natural numbers. This could be defined as a boolean function, as we have seen, but we can also define it inductively

```
data _≤_ : Nat -> Nat -> Set where
  leq-zero : {n : Nat} -> zero ≤ n
  leq-suc  : {m n : Nat} -> m ≤ n -> suc m ≤ suc n
```

One advantage of this approach is that we can pattern match on the proof object. This makes proving properties of `_≤_` easier. For instance,

```
leq-trans : {l m n : Nat} -> l ≤ m -> m ≤ n -> l ≤ n
leq-trans leq-zero    _           = leq-zero
leq-trans (leq-suc p) (leq-suc q) = leq-suc (leq-trans p q)
```

### 2.6 More on pattern matching

We have seen how to pattern match on the arguments of a function, but sometimes you want to pattern match on the result of some intermediate computation. In Haskell and ML this is done on the right hand side using a case or match expression. However, as we have learned, when pattern matching on an expression in a dependently typed language, you not only learn something about the shape of the expression, but you can also learn things about other expressions. For instance, pattern matching on an expression of type `Vec A n` will reveal information about `n`. This is not captured by the usual case expression, so instead of a case expression Agda provides a way of matching on intermediate computations on the left hand side.

**The *with* construct** The idea is that if you want to pattern match on an expression `e` in the definition of a function `f`, you abstract `f` over the value of `e`, effectively adding another argument to `f` which can then be matched on in the usual fashion. This abstraction is performed by the `with` construct. For instance,

```
min : Nat -> Nat -> Nat
min x y with x < y
min x y | true  = x
min x y | false = y
```

The equations for `min` following the with abstraction have an extra argument, separated from the original arguments by a vertical bar, corresponding to the value of the expression `x < y`. You can abstract over multiple expressions at the same time, separating them by vertical bars and you can nest with abstractions. In the left hand side, with abstracted arguments should be separated by vertical bars.

In this case pattern matching on `x < y` doesn't tell us anything interesting about the arguments of `min`, so repeating the left hand sides is a bit tedious. When this is the case you can replace the left hand side with `...`:

```
filter : {A : Set} -> (A -> Bool) -> List A -> List A
filter p [] = []
filter p (x :: xs) with p x
... | true  = x :: filter p xs
... | false = filter p xs
```

Here is an example when we do learn something interesting. Given two numbers we can compare them to see if they are equal. Rather than returning an uninteresting boolean, we can return a proof that the numbers are indeed equal when this is the case, and an explanation of why they are different when this is the case:

```
data _≠_ : Nat -> Nat -> Set where
  z≠s : {n : Nat} -> zero ≠ suc n
  s≠z : {n : Nat} -> suc n ≠ zero
  s≠s : {m n : Nat} -> m ≠ n -> suc m ≠ suc n

data Equal? (n m : Nat) : Set where
  eq  : n == m -> Equal? n m
  neq : n ≠ m -> Equal? n m
```

Two natural numbers are different if one is `zero` and the other `suc` of something, or if both are successors but their predecessors are different. Now we can define the function `equal?` to check if two numbers are equal:

```
equal? : (n m : Nat) -> Equal? n m
equal? zero    zero    = eq refl
equal? zero    (suc m) = neq z≠s
equal? (suc n) zero    = neq s≠z
equal? (suc n) (suc m) with equal? n m
equal? (suc n) (suc .n) | eq refl = eq refl
equal? (suc n) (suc m)  | neq p   = neq (s≠s p)
```

Note that in the case where both numbers are successors we learn something by pattern matching on the proof that the predecessors are equal. We will see more examples of this kind of informative datatypes in Section 3.1.

When you abstract over an expression using `with`, that expression is abstracted from the entire context. This means that if the expression occurs in the type of an argument to the function or in the result type, this occurrence will be replaced by the with-argument on the left hand side. For example, suppose we want to prove something about the `filter` function. That the only thing it does is throwing away some elements of its argument, say. We can define what it means for one list to be a sublist of another list:

```
infix 20 _⊆_
data _⊆_ {A : Set} : List A -> List A -> Set where
  stop : [] ⊆ []
  drop : forall {xs y ys} -> xs ⊆ ys ->      xs ⊆ y :: ys
  keep : forall {x xs ys} -> xs ⊆ ys -> x :: xs ⊆ x :: ys
```

The intuition is that to obtain a sublist of a given list, each element can either be dropped or kept. When the type checker can figure out the type of an argument in a function type you can use the `forall` syntax:

– `forall {x y} a b -> A` is short for `{x : _}{y : _}(a : _)(b : _) -> A`.

Using this definition we can prove that `filter` computes a sublist of its argument:

```
lem-filter : {A : Set}(p : A -> Bool)(xs : List A) ->
             filter p xs ⊆ xs
lem-filter p []        = stop
lem-filter p (x :: xs) with p x
... | true  = keep (lem-filter p xs)
... | false = drop (lem-filter p xs)
```

The interesting case is the `_::_` case. Let us walk through it slowly:

```
-- lem-filter p (x :: xs) = ?
```

At this point the goal that we have to prove is

```
-- (filter p (x :: xs) | p x) ⊆ x :: xs
```

In the goal `filter` has been applied to its with abstracted argument `p x` and will not reduce any further. Now, when we abstract over `p x` it will be abstracted from the goal type so we get

```
-- lem-filter p (x :: xs) with p x
-- ... | px = ?
```

where `p x` has been replaced by `px` in the goal type

```
-- (filter p (x :: xs) | px) ⊆ x :: xs
```

Now, when we pattern match on `px` the call to `filter` will reduce and we get

```
-- lem-filter p (x :: xs) with p x
-- ... | true  = ?  {- x :: filter p xs ⊆ x :: xs -}
-- ... | false = ?  {-      filter p xs ⊆ x :: xs -}
```

In some cases, it can be helpful to use `with` to abstract over an expression which you are not going to pattern match on. In particular, if you expect this expression to be instantiated by pattern matching on something else. Consider the proof that `n + zero == n`:

```
lem-plus-zero : (n : Nat) -> n + zero == n
lem-plus-zero zero = refl
lem-plus-zero (suc n) with n + zero | lem-plus-zero n
... | .n | refl = refl
```

In the step case we would like to pattern match on the induction hypothesis `n + zero == n` in order to prove `suc n + zero == suc n`, but since `n + zero` cannot be unified with `n` that is not allowed. However, if we abstract over `n + zero`, calling it `m`, we are left with the induction hypothesis `m == n` and the goal `suc m == suc n`. Now we can pattern match on the induction hypothesis, instantiating `m` to `n`.

### 2.7 Modules

The module system in Agda is primarily used to manage name spaces. In a dependently typed setting you could imagine having modules as first class objects that could be passed around and created on the fly, but in Agda this is not the case.

We have already seen that each file must define a single top-level module containing all the declarations in the file. These declarations can in turn be modules.

```
module M where
  data Maybe (A : Set) : Set where
    nothing : Maybe A
    just    : A -> Maybe A

  maybe : {A B : Set} -> B -> (A -> B) -> Maybe A -> B
  maybe z f nothing  = z
  maybe z f (just x) = f x
```

By default all names declared in a module are visible from the outside. If you want to hide parts of a module you can declare it `private`:

```
module A where
  private
    internal : Nat
    internal = zero

  exported : Nat -> Nat
  exported n = n + internal
```

To access public names from another module you can qualify the name by the name of the module.

```
mapMaybe₁ : {A B : Set} -> (A -> B) -> M.Maybe A -> M.Maybe B
mapMaybe₁ f M.nothing  = M.nothing
mapMaybe₁ f (M.just x) = M.just (f x)
```

Modules can also be opened, locally or on top-level:

```
mapMaybe₂ : {A B : Set} -> (A -> B) -> M.Maybe A -> M.Maybe B
mapMaybe₂ f m = let open M in maybe nothing (just ∘ f) m

open M

mapMaybe₃ : {A B : Set} -> (A -> B) -> Maybe A -> Maybe B
mapMaybe₃ f m = maybe nothing (just ∘ f) m
```

When opening a module you can control which names are brought into scope with the `using`, `hiding`, and `renaming` keywords. For instance, to open the `Maybe` module without exposing the `maybe` function, and using different names for the type and the constructors we can say

```
open M hiding (maybe)
           renaming (Maybe to _option; nothing to none; just to some)

mapOption : {A B : Set} -> (A -> B) -> A option -> B option
mapOption f none     = none
mapOption f (some x) = some (f x)
```

Renaming is just cosmetic, `Maybe A` and `A option` are interchangable.

```
mtrue : Maybe Bool
mtrue = mapOption not (just false)
```

**Parameterised modules** Modules can be parameterised by arbitrary types[8].

```
module Sort (A : Set)(_<_ : A -> A -> Bool) where
  insert : A -> List A -> List A
  insert y [] = y :: []
  insert y (x :: xs) with x < y
  ... | true  = x :: insert y xs
  ... | false = y :: x :: xs

  sort : List A -> List A
  sort []        = []
  sort (x :: xs) = insert x (sort xs)
```

When looking at the functions in parameterised module from the outside, they take the module parameters as arguments, so

```
sort₁ : (A : Set)(_<_ : A -> A -> Bool) -> List A -> List A
sort₁ = Sort.sort
```

You can apply the functions in a parameterised module to the module parameters all at once, by instantiating the module

```
module SortNat = Sort Nat _<_
```

This creates a new module `SortNat` with functions `insert` and `sort`.

```
sort₂ : List Nat -> List Nat
sort₂ = SortNat.sort
```

Often you want to instantiate a module and open the result, in which case you can simply write

```
open Sort Nat _<_ renaming (insert to insertNat; sort to sortNat)
```

without having to give a name to the instantiated module.

Sometimes you want to export the contents of another module from the current module. In this case you can open the module *publicly* using the `public` keyword:

```
module Lists (A : Set)(_<_ : A -> A -> Bool) where
  open Sort A _<_ public
  minimum : List A -> Maybe A
  minimum xs with sort xs
  ... | []      = nothing
  ... | y :: ys = just y
```

Now the `Lists` module will contain `insert` and `sort` as well as the `minimum` function.

[8] But not by other modules.

**Importing modules from other files** Agda programs can be split over multiple files. To use definitions from a module defined in another file the module has to be *imported*. Modules are imported by their names, so if you have a module `A.B.C` in a file `/some/local/path/A/B/C.agda` it is imported with the statement `import A.B.C`. In order for the system to find the file `/some/local/path` must be in Agda's search path.[9].

I have a file `Logic.agda` in the same directory as these notes, defining logical conjunction and disjunction. To import it we say

```
import Logic using (_∧_; _∨_)
```

Note that you can use the same namespace control keywords as when opening modules. Importing a module does not automatically open it (like when you say `import qualified` in Haskell). You can either open it separately with an open statement, or use the short form `open import Logic`.

Splitting a program over several files will improve type checking performance, since when you are making changes the type checker only has to type check the files that are influenced by the changes.

### 2.8 Records

We have seen a record type already, namely the record type with no fields which was used to model the true proposition. Now let us look at record types with fields. A record type is declared much like a datatype where the fields are indicated by the `field` keyword. For instance

```
record Point : Set where
  field x : Nat
        y : Nat
```

This declares a record type `Point` with two natural number fields `x` and `y`. To construct an element of `Point` you write

```
mkPoint : Nat -> Nat -> Point
mkPoint a b = record{ x = a; y = b }
```

To allow projection of the fields from a record, each record type comes with a module of the same name. This module is parameterised by an element of the record type and contains projection functions for the fields. In the point example we get a module

```
-- module Point (p : Point) where
--   x : Nat
--   y : Nat
```

[9] The search path can be set from emacs by executing `M-x customize-group agda2`.

This module can be used as it is or instantiated to a particular record.

```
getX : Point -> Nat
getX = Point.x

abs² : Point -> Nat
abs² p = let open Point p in x * x + y * y
```

At the moment you cannot pattern match on records, but this will hopefully be possible in a later version of Agda.

It is possible to add your own functions to the module of a record by including them in the record declaration after the fields.

```
record Monad (M : Set -> Set) : Set1 where
  field
    return : {A : Set} -> A -> M A
    _>>=_  : {A B : Set} -> M A -> (A -> M B) -> M B

  mapM : {A B : Set} -> (A -> M B) -> List A -> M (List B)
  mapM f [] = return []
  mapM f (x :: xs) = f x       >>= \y  ->
                     mapM f xs >>= \ys ->
                     return (y :: ys)

mapM' : {M : Set -> Set} -> Monad M ->
        {A B : Set} -> (A -> M B) -> List A -> M (List B)
mapM' Mon f xs = Monad.mapM Mon f xs
```

## 2.9 Exercises

*Exercise 2.1.* Matrix transposition

We can model an $n \times m$ matrix as a vector of vectors:

```
Matrix : Set -> Nat -> Nat -> Set
Matrix A n m = Vec (Vec A n) m
```

The goal of this exercise is to define the transposition of such a matrix.

(a) Define a function to compute a vector containing $n$ copies of an element $x$.

```
vec : {n : Nat}{A : Set} -> A -> Vec A n
vec {n} x = {! !}
```

(b) Define point-wise application of a vector of functions to a vector of arguments.

```
infixl 90 _$_
_$_ : {n : Nat}{A B : Set} -> Vec (A -> B) n -> Vec A n -> Vec B n
fs $ xs = {! !}
```

(c) Define matrix transposition in terms of these two functions.

```
transpose : forall {A n m} -> Matrix A n m -> Matrix A m n
transpose xss = {! !}
```

*Exercise 2.2.* Vector lookup

Remember `tabulate` and `_!_` from Section 2.4. Prove that they are indeed each other's inverses.

(a) This direction should be relatively easy.

```
lem-!-tab : forall {A n} (f : Fin n -> A)(i : Fin n) ->
            tabulate f ! i == f i
lem-!-tab f i = {! !}
```

(b) This direction might be trickier.

```
lem-tab-! : forall {A n} (xs : Vec A n) -> tabulate (_!_ xs) == xs
lem-tab-! xs = {! !}
```

*Exercise 2.3.* Sublists

Remember the representation of sublists from Section 2.4:

```
data _⊆_ {A : Set} : List A -> List A -> Set where
  stop : [] ⊆ []
  drop : forall {x xs ys} -> xs ⊆ ys ->      xs ⊆ x :: ys
  keep : forall {x xs ys} -> xs ⊆ ys -> x :: xs ⊆ x :: ys
```

(a) Prove the reflexivity and transitivity of `_⊆_`:

```
⊆-refl : {A : Set}{xs : List A} -> xs ⊆ xs
⊆-refl {xs = xs} = {! !}

⊆-trans : {A : Set}{xs ys zs : List A} ->
          xs ⊆ ys -> ys ⊆ zs -> xs ⊆ zs
⊆-trans p q = {! !}
```

Instead of defining the sublist relation we can define the type of sublists of a given list as follows:

```
infixr 30 _::_
data SubList {A : Set} : List A -> Set where
  []   : SubList []
  _::_ : forall x {xs} -> SubList xs -> SubList (x :: xs)
  skip : forall {x xs} -> SubList xs -> SubList (x :: xs)
```

(b) Define a function to extract the list corresponding to a sublist.

```
forget : {A : Set}{xs : List A} -> SubList xs -> List A
forget s = {! !}
```

(c) Now, prove that a `SubList` is a sublist in the sense of $\_\subseteq\_$.

```
lem-forget : {A : Set}{xs : List A}(zs : SubList xs) ->
             forget zs ⊆ xs
lem-forget zs = {! !}
```

(d) Give an alternative definition of filter which satisfies the sublist property by construction.

```
filter' : {A : Set} -> (A -> Bool) -> (xs : List A) -> SubList xs
filter' p xs = {! !}
```

(e) Define the complement of a sublist

```
complement : {A : Set}{xs : List A} -> SubList xs -> SubList xs
complement zs = {! !}
```

(f) Compute all sublists of a given list

```
sublists : {A : Set}(xs : List A) -> List (SubList xs)
sublists xs = {! !}
```

## 3 Programming Techniques

In this section we will describe and exemplify a couple of programming techniques which are made available in dependently typed languages: *views* and *universe constructions*.

### 3.1 Views

As we have seen pattern matching in Agda can reveal information not only about the term being matched but also about terms occurring in the type of this term. For instance, matching a proof of `x == y` against the `refl` constructor we (and the type checker) will learn that `x` and `y` are the same.

We can exploit this, and design datatypes whose sole purpose is to tell us something interesting about its indices. We call such a datatype a *view* [5]. To use the view we define a view function, computing an element of the view for arbitrary indices.

This section on views is defined in the file `Views.lagda` so here is the top-level module declaration:

```
module Views where
```

**Natural number parity** Let us start with an example. We all know that any natural number $n$ can be written on the form $2k$ or $2k + 1$ for some $k$. Here is a view datatype expressing that. We use the natural numbers defined in the summer school library [7] module `Data.Nat`.

```
open import Data.Nat

data Parity : Nat -> Set where
  even : (k : Nat) -> Parity (k * 2)
  odd  : (k : Nat) -> Parity (1 + k * 2)
```

An element of `Parity n` tells you if `n` is even or odd, i.e. if $n = 2k$ or $n = 2k + 1$, and in each case what $k$ is. The reason for writing `k * 2` and `1 + k * 2` rather than `2 * k` and `2 * k + 1` has to do with the fact that `_+_` and `_*_` are defined by recursion over their first argument. This way around we get a better reduction behaviour.

Now, just defining the view datatype isn't very helpful. We also need to show that any natural number can be viewed in this way. In other words, that given an arbitrary natural number `n` we can to compute an element of `Parity n`.

```
parity : (n : Nat) -> Parity n
parity zero = even zero
parity (suc n) with parity n
parity (suc .(k * 2))     | even k = odd k
parity (suc .(1 + k * 2)) | odd  k = even (suc k)
```

In the `suc n` case we use the view recursively to find out the parity of `n`. If `n = k * 2` then `suc n = 1 + k * 2` and if `n = 1 + k * 2` then `suc n = suc k * 2`.

In effect, this view gives us the ability to pattern match on a natural number with the patterns `k * 2` and `1 + k * 2`. Using this ability, defining the function that divides a natural number by two is more or less trivial:

```
half : Nat -> Nat
half n with parity n
half .(k * 2)     | even k = k
half .(1 + k * 2) | odd k  = k
```

Note that `k` is bound in the pattern for the view, not in the dotted pattern for the natural number.

**Finding an element in a list** Let us turn our attention to lists. First some imports: we will use the definitions of lists and booleans from the summer school library [7].

```
open import Data.Function
open import Data.List
open import Data.Bool
```

Now, given a predicate `P` and a list `xs` we can define what it means for `P` to hold for all elements of `xs`:

```
infixr 30 _:all:_
data All {A : Set}(P : A -> Set) : List A -> Set where
  all[]   : All P []
  _:all:_ : forall {x xs} -> P x -> All P xs -> All P (x :: xs)
```

A proof of `All P xs` is simply a list of proofs of `P x` for each element `x` of `xs`. Note that `P` does not have to be a decidable predicate. To turn a decidable predicate into a general predicate we define a function `satisfies`.

```
satisfies : {A : Set} -> (A -> Bool) -> A -> Set
satisfies p x = isTrue (p x)
```

Using the `All` datatype we could prove the second part of the correctness of the `filter` function, namely that all the elements of the result of `filter` satisfies the predicate: `All (satisfies p) (filter p xs)`. This is left as an exercise. Instead, let us define some interesting views on lists.

Given a decidable predicate on the elements of a list, we can either find an element in the list that satisfies the predicate, or else all elements satifies the negation of the predicate. Here is the corresponding view datatype:

```
data Find {A : Set}(p : A -> Bool) : List A -> Set where
  found : (xs : List A)(y : A) -> satisfies p y -> (ys : List A) ->
          Find p (xs ++ y :: ys)
  not-found : forall {xs} -> All (satisfies (not ∘ p)) xs ->
              Find p xs
```

We don't specify which element to use as a witness in the `found` case. If we wanted the view to always return the first (or last) matching element we could force the elements of `xs` (or `ys`) to satisfy the negation of `p`. To complete the view we need to define the view function computing an element of `Find p xs` for any `p` and `xs`. Here is a first attempt:

```
find₁ : {A : Set}(p : A -> Bool)(xs : List A) -> Find p xs
find₁ p [] = not-found all[]
find₁ p (x :: xs) with p x
... | true  = found [] x {! !} xs
... | false = {! !}
```

In the case where `p x` is `true` we want to return `found` (hence, returning the first match), but there is a problem. The type of the hole (`{! !}`) is `isTrue (p x)`, even though we already matched on `p x` and found out that it was `true`. The problem is that when we abstracted over `p x` we didn't know that we wanted to use the `found` constructor, so there were no `p x` to abstract over. Remember that `with` doesn't remember the connection between the with-term and the patterns. One solution to this problem is to make this connection explicit with a proof object. The idea is to not abstract over the term itself but rather over an arbitrary term of the same type and a proof that it is equal to the original term. Remember the type of equality proofs:

```
data _==_ {A : Set}(x : A) : A -> Set where
  refl : x == x
```

Now we define the type of elements of a type `A` together with proofs that they are equal to some given `x` in `A`.

```
data Inspect {A : Set}(x : A) : Set where
  it : (y : A) -> x == y -> Inspect x
```

There is one obvious way to construct an element of `Inspect x`, namely to pick `x` as the thing which is equal to `x`.

```
inspect : {A : Set}(x : A) -> Inspect x
inspect x = it x refl
```

We can now define `find` by abstracting over `inspect (p x)` rather than `p x` itself. This will provide us with either a proof of `p x == true` or a proof of `p x == false` which we can use in the arguments to `found` and `not-found`. First we need a couple of lemmas about `isTrue` and `isFalse`:

```
trueIsTrue : {x : Bool} -> x == true -> isTrue x
trueIsTrue refl = _

falseIsFalse : {x : Bool} -> x == false -> isFalse x
falseIsFalse refl = _
```

Now we can define `find` without any problems.

```
find : {A : Set}(p : A -> Bool)(xs : List A) -> Find p xs
find p [] = not-found all[]
find p (x :: xs) with inspect (p x)
... | it true prf = found [] x (trueIsTrue prf) xs
... | it false prf with find p xs
find p (x :: ._) | it false _   | found xs y py ys =
  found (x :: xs) y py ys
find p (x :: xs) | it false prf | not-found npxs =
  not-found (falseIsFalse prf :all: npxs)
```

In the case where `p x` is true, `inspect (p x)` matches `it true prf` where `prf : p x == true`. Using our lemma we can turn this into the proof of `isTrue (p x)` that we need for the third argument of `found`. We get a similar situation when `p x` is false and `find p xs` returns `not-found`.

**Indexing into a list** In Sections 2.4 and Section 2.5 we saw two ways of safely indexing into a list. In both cases the type system guaranteed that the index didn't point outside the list. However, sometimes we have no control over the value of the index and it might well be that it is pointing outside the list. One solution in this case would be to wrap the result of the lookup function in a maybe type, but maybe types don't really tell you anything very interesting and we can do a lot better. First let us define the type of proofs that an element `x` is in a list `xs`.

```
data _∈_ {A : Set}(x : A) : List A -> Set where
  hd : forall {xs}   -> x ∈ x :: xs
  tl : forall {y xs} -> x ∈ xs -> x ∈ y :: xs
```

The first element of a list is a member of the list, and any element of the tail of a list is also an element of the entire list. Given a proof of `x ∈ xs` we can compute the index at which `x` occurs in `xs` simply by counting the number of `tl`s in the proof.

```
index : forall {A}{x : A}{xs} -> x ∈ xs -> Nat
index hd     = zero
index (tl p) = suc (index p)
```

Now, let us define a view on natural numbers `n` with respect to a list `xs`. Either `n` indexes some `x` in `xs` in which case it is of the form `index p` for some proof `p : x ∈ xs`, or `n` points outside the list, in which case it is of the form `length xs + m` for some `m`.

```
data Lookup {A : Set}(xs : List A) : Nat -> Set where
  inside  : (x : A)(p : x ∈ xs) -> Lookup xs (index p)
  outside : (m : Nat) -> Lookup xs (length xs + m)
```

In the case that `n` is a valid index we not only get the element at the corresponding position in `xs` but we are guaranteed that this is the element that is returned. There is no way a lookup function could cheat and always return the first element, say. In the case that `n` is indexing outside the list we also get some more information. We get a proof that `n` is out of bounds and we also get to know by how much.

Defining the lookup function is no more difficult than it would have been to define the lookup function returning a maybe.

```
_!_ : {A : Set}(xs : List A)(n : Nat) -> Lookup xs n
[] ! n = outside n
(x :: xs) ! zero  = inside x hd
(x :: xs) ! suc n with xs ! n
(x :: xs) ! suc .(index p)       | inside y p = inside y (tl p)
(x :: xs) ! suc .(length xs + n) | outside n  = outside n
```

**A type checker for λ-calculus** To conclude this section on views, let us look at a somewhat bigger example: a type checker for simply typed $\lambda$-calculus. This example is due to Conor McBride [5] and was first implemented in Epigram. His version not only guaranteed that when the type checker said ok things were really ok, but also provided a detailed explanation in the case where type checking failed. We will focus on the positive side here and leave the reporting of sensible and guaranteed precise error message as an exercise.

First, let us define the type language. We have one base type `ı` and a function type.

```
infixr 30 _⇒_
data Type : Set where
  ı   : Type
  _⇒_ : Type -> Type -> Type
```

When doing type checking we will inevitably have to compare types for equality, so let us define a view.

```
data Equal? : Type -> Type -> Set where
  yes : forall {τ} -> Equal? τ τ
  no  : forall {σ τ} -> Equal? σ τ

_=?=_ : (σ τ : Type) -> Equal? σ τ
ı         =?= ı       = yes
ı         =?= (_ ⇒ _) = no
(_ ⇒ _)   =?= ı       = no
(σ₁ ⇒ τ₁) =?= (σ₂ ⇒ τ₂) with σ₁ =?= σ₂ | τ₁ =?= τ₂
(σ ⇒ τ)   =?= (.σ ⇒ .τ) | yes | yes = yes
(σ₁ ⇒ τ₁) =?= (σ₂ ⇒ τ₂) | _   | _   = no
```

Note that we don't give any justification in the `no` case. The `_=?=_` could return `no` all the time without complaints from the type checker. In the `yes` case, however, we guarantee that the two types are identical.

Next up we define the type of raw lambda terms. We use unchecked deBruijn indices to represent variables.

```
infixl 80 _$_
data Raw : Set where
  var : Nat -> Raw
  _$_ : Raw -> Raw -> Raw
  lam : Type -> Raw -> Raw
```

We use Church style terms in order to simplify type inference. The idea with our type checker is that it should take a raw term and return a well-typed term, so we need to define the type of well-typed $\lambda$-terms with respect to a context $\Gamma$ and a type $\tau$.

```
Cxt = List Type

data Term (Γ : Cxt) : Type -> Set where
  var : forall {τ} -> τ ∈ Γ -> Term Γ τ
  _$_ : forall {σ τ} -> Term Γ (σ ⇒ τ) -> Term Γ σ -> Term Γ τ
  lam : forall σ {τ} -> Term (σ :: Γ) τ -> Term Γ (σ ⇒ τ)
```

We represent variables by proofs that a type is in the context. Remember that the proofs of list membership provide us with an index into the list where the element can be found. Given a well-typed term we can erase all the type information and get a raw term.

```
erase : forall {Γ τ} -> Term Γ τ -> Raw
erase (var x)   = var (index x)
erase (t $ u)   = erase t $ erase u
erase (lam σ t) = lam σ (erase t)
```

In the variable case we turn the proof into a natural number using the `index` function.

Now we are ready to define the view of a raw term as either being the erasure of a well-typed term or not. Again, we don't provide any justification for giving a negative result. Since, we are doing type inference the type is not a parameter of the view but computed by the view function.

```
data Infer (Γ : Cxt) : Raw -> Set where
  ok  : (τ : Type)(t : Term Γ τ) -> Infer Γ (erase t)
  bad : {e : Raw} -> Infer Γ e
```

The view function is the type inference function taking a raw term and computing an element of the `Infer` view.

```
infer : (Γ : Cxt)(e : Raw) -> Infer Γ e
```

Let us walk through the three cases: variable, application, and lambda abstraction.

```
infer Γ (var n)   with Γ ! n
infer Γ (var .(length Γ + n)) | outside n  = bad
infer Γ (var .(index x))      | inside σ x = ok σ (var x)
```

In the variable case we need to take case of the fact that the raw variable might be out of scope. We can use the lookup function `_!_` we defined above for that. When the variable is in scope the lookup function provides us with the type of the variable and the proof that it is in scope.

```
infer Γ (e₁ $ e₂)
  with infer Γ e₁
infer Γ (e₁ $ e₂)          | bad     = bad
infer Γ (.(erase t₁) $ e₂) | ok ι t₁ = bad
infer Γ (.(erase t₁) $ e₂) | ok (σ ⇒ τ) t₁
    with infer Γ e₂
infer Γ (.(erase t₁) $ e₂) | ok (σ ⇒ τ) t₁ | bad = bad
infer Γ (.(erase t₁) $ .(erase t₂)) | ok (σ ⇒ τ) t₁ | ok σ' t₂
      with σ =?= σ'
infer Γ (.(erase t₁) $ .(erase t₂))
  | ok (σ ⇒ τ) t₁ | ok .σ t₂ | yes = ok τ (t₁ $ t₂)
infer Γ (.(erase t₁) $ .(erase t₂))
  | ok (σ ⇒ τ) t₁ | ok σ' t₂ | no = bad
```

The application case is the bulkiest simply because there are a lot of things we need to check: that the two terms are type correct, that the first term has a function type and that the type of the second term matches the argument type of the first term. This is all done by pattern matching on recursive calls to the `infer` view and the type equality view.

```
infer Γ (lam σ e) with infer (σ :: Γ) e
infer Γ (lam σ .(erase t)) | ok τ t  = ok (σ ⇒ τ) (lam σ t)
infer Γ (lam σ e)          | bad     = bad
```

Finally, the lambda case is very simple. If the body of the lambda is type correct in the extended context, then the lambda is well-typed with the corresponding function type.

Without much effort we have defined a type checker for simply typed $\lambda$-calculus that not only is guaranteed to compute well-typed terms, but also guarantees that the erasure of the well-typed term is the term you started with.

### 3.2 Universes

The second programming technique we will look at that is not available in non-dependently typed languages is *universe construction*. First the module header.

```
module Universes where
```

A universe is a set of types (or type formers) and a universe construction consists of a type of codes and a decoding function mapping codes to types in the universe. The purpose of a universe construction is to be able to define functions over the types of the universe by inspecting their codes. In fact we have seen an example of a universe construction already.

**A familiar universe** The universe of decidable propositions consists of the singleton type `True` and the empty type `False`. Codes are booleans and the decoder is the `isTrue` function.

```
data   False : Set where
record True  : Set where

data Bool : Set where
  true  : Bool
  false : Bool

isTrue : Bool -> Set
isTrue true  = True
isTrue false = False
```

Now functions over decidable propositions can be defined by manipulating the boolean codes. For instance, we can define negation and conjunction as functions on codes and prove some properties of the corresponding propositions.

```
infix  30 not_
infixr 25 _and_

not_ : Bool -> Bool
not true  = false
not false = true

_and_ : Bool -> Bool -> Bool
true  and x = x
false and _ = false

notNotId : (a : Bool) -> isTrue (not not a) -> isTrue a
```

```
notNotId true  p = p
notNotId false ()

andIntro : (a b : Bool) -> isTrue a -> isTrue b -> isTrue (a and b)
andIntro true  _ _  p = p
andIntro false _ () _
```

A nice property of this universe is that proofs of `True` can be found automatically. This means that if you have a function taking a proof of a precondition as an argument, where you expect the precondition to be trivially true at the point where you are calling the function, you can make the precondition an implicit argument. For instance, if you expect to mostly divide by concrete numbers, division of natural numbers can be given the type signature

```
open import Data.Nat

nonZero : Nat -> Bool
nonZero zero    = false
nonZero (suc _) = true

postulate _div_ : Nat -> (m : Nat){p : isTrue (nonZero m)} -> Nat

three = 16 div 5
```

Here the proof obligation `isTrue (nonZero 5)` will reduce to `True` and solved automatically by the type checker. Note that if you tell the type checker that you have defined the type of natural numbers, you are allowed to use natural number literals like `16` and `5`. This has been done in the library.

**Universes for generic programming** Generic programming deals with the problem of defining functions generically over a set of types. We can achieve this by defining a universe for the set of types we are interested in. Here is a simple example of how to program generically over the set of types computed by fixed points over polynomial functors.

First we define a type of codes for polynomial functors.

```
data Functor : Set1 where
  |Id|  : Functor
  |K|   : Set -> Functor
  _|+|_ : Functor -> Functor -> Functor
  _|x|_ : Functor -> Functor -> Functor
```

A polynomial functor is either the identity functor, a constant functor, the disjoint union of two functors, or the cartesian product of two functors.

Since codes for functors can contain arbitrary `Set`s (in the case of the constant functor) the type of codes cannot itself be a `Set`, but lives in `Set1`.

Before defining the decoding function for functors we define datatypes for disjoint union and cartesian product.

```
data _⊕_ (A B : Set) : Set where
  inl : A -> A ⊕ B
  inr : B -> A ⊕ B

data _×_ (A B : Set) : Set where
  _,_ : A -> B -> A × B

infixr 50 _|+|_ _⊕_
infixr 60 _|x|_ _×_
```

The decoding function takes a code for a functor to a function on `Set`s and is computed recursively over the code.

```
[_] : Functor -> Set -> Set
[ |Id|    ] X = X
[ |K| A   ] X = A
[ F |+| G ] X = [ F ] X ⊕ [ G ] X
[ F |x| G ] X = [ F ] X × [ G ] X
```

Since it's called a functor it ought to support a map operation. We can define this by recursion over the code.

```
map : (F : Functor){X Y : Set} -> (X -> Y) -> [ F ] X -> [ F ] Y
map |Id|      f x       = f x
map (|K| A)   f c       = c
map (F |+| G) f (inl x) = inl (map F f x)
map (F |+| G) f (inr y) = inr (map G f y)
map (F |x| G) f (x , y) = map F f x , map G f y
```

Next we define the least fixed point of a polynomial functor.

```
data μ_ (F : Functor) : Set where
  <_> : [ F ] (μ F) -> μ F
```

To ensure termination, recursive datatypes must be strictly positive and this is checked by the type checker. Our definition of least fixed point goes through, since the type checker can spot that `[_]` is strictly positive in its second argument.

With this definition we can define a generic fold operation on least fixed points. Grabbing for the closest category theory text book we might try something like this

```
-- fold : (F : Functor){A : Set} -> ([ F ] A -> A) -> μ F -> A
-- fold F φ < x > = φ (map F (fold F φ) x)
```

Unfortunately, this definition does not pass the termination checker since the recursive call to `fold` is passed to the higher order function `map` and the termination checker cannot see that `map` isn't applying it to bad things.

To make `fold` pass the termination checker we can fuse `map` and `fold` into a single function `mapFold F G φ x = map F (fold G φ) x` defined recursively over `x`. We need to keep two copies of the functor since `fold` is always called on the same functor, whereas `map` is defined by taking its functor argument apart.

```
mapFold : forall {X} F G -> ([ G ] X -> X) -> [ F ] (μ G) -> [ F ] X
mapFold |Id|        G φ < x >   = φ (mapFold G G φ x)
mapFold (|K| A)     G φ c       = c
mapFold (F₁ |+| F₂) G φ (inl x) = inl (mapFold F₁ G φ x)
mapFold (F₁ |+| F₂) G φ (inr y) = inr (mapFold F₂ G φ y)
mapFold (F₁ |x| F₂) G φ (x , y) = mapFold F₁ G φ x , mapFold F₂ G φ y

fold : {F : Functor}{A : Set} -> ([ F ] A -> A) -> μ F -> A
fold {F} φ < x > = φ (mapFold F F φ x)
```

There is a lot more fun to be had here, but let us make do with a couple of examples. Both natural numbers and lists are examples of least fixed points of polynomial functors:

```
NatF = |K| True |+| |Id|
NAT  = μ NatF

Z : NAT
Z = < inl _ >

S : NAT -> NAT
S n = < inr n >

ListF = \A -> |K| True |+| |K| A |x| |Id|
LIST  = \A -> μ (ListF A)

nil : {A : Set} -> LIST A
nil = < inl _ >

cons : {A : Set} -> A -> LIST A -> LIST A
cons x xs = < inr (x , xs) >
```

To make implementing the argument to fold easier we introduce a few helper functions:

```
[_||_] : {A B C : Set} -> (A -> C) -> (B -> C) -> A ⊕ B -> C
[ f || g ] (inl x) = f x
[ f || g ] (inr y) = g y

uncurry : {A B C : Set} -> (A -> B -> C) -> A × B -> C
uncurry f (x , y) = f x y

const : {A B : Set} -> A -> B -> A
const x y = x
```

Finally some familiar functions expressed as folds.

```
foldr : {A B : Set} -> (A -> B -> B) -> B -> LIST A -> B
foldr {A}{B} f z = fold [ const z || uncurry f ]

plus : NAT -> NAT -> NAT
plus n m = fold [ const m || S ] n
```

**Universes for overloading** At the moment, Agda does not have a class system like the one in Haskell. However, a limited form of overloading can be achieved using universes. The idea is simply if you know in advance at which types you want to overload a function, you can construct a universe for these types and define the overloaded function by pattern matching on a code.

A simple example: suppose we want to overload equality for some of our standard types. We start by defining our universe:

```
open import Data.List

data Type : Set where
  bool : Type
  nat  : Type
  list : Type -> Type
  pair : Type -> Type -> Type

El : Type -> Set
El nat        = Nat
El bool       = Bool
El (list a)   = List (El a)
El (pair a b) = El a × El b
```

In order to achieve proper overloading it is important that we don't have to supply the code explicitly everytime we are calling the overloaded function. In this case we won't have to since the decoding function computes distinct datatypes in each clause. This means that the type checker can figure out a code from its decoding. For instance, the only code that

can decode into `Bool` is `bool`, and if the decoding of a code is a product type then the code must be `pair` of some codes.

Now an overloaded equality function simply takes an implicit code and computes a boolean relation over the semantics of the code.

```
infix 30 _==_
_==_ : {a : Type} -> El a -> El a -> Bool

_==_ {nat} zero    zero    = true
_==_ {nat} (suc _) zero    = false
_==_ {nat} zero    (suc _) = false
_==_ {nat} (suc n) (suc m) = n == m

_==_ {bool} true  x = x
_==_ {bool} false x = not x

_==_ {list a} [] []       = true
_==_ {list a} (_ :: _) [] = false
_==_ {list a} [] (_ :: _) = false
_==_ {list a} (x :: xs) (y :: ys) = x == y and xs == ys

_==_ {pair a b} (x₁ , y₁) (x₂ , y₂) = x₁ == x₂ and y₁ == y₂
```

In the recursive calls of `_==_` the code argument is inferred automatically. The same happens when we use our equality function on concrete examples:

```
example₁ : isTrue (2 + 2 == 4)
example₁ = _

example₂ : isTrue (not (true :: false :: [] == true :: true :: []))
example₂ = _
```

In summary, universe constructions allows us to define functions by pattern matching on (codes for) types. We have seen a few simple examples, but there are a lot of other interesting possibilities. For example

- XML schemas as codes for the types of well-formed XML documents,
- a universe of tables in a relational database, allowing us to make queries which are guaranteed to be well-typed

### 3.3 Exercises

*Exercise 3.1.* Natural numbers

Here is a view on pairs of natural numbers.

```
data Compare : Nat -> Nat -> Set where
  less : forall {n} k -> Compare n (n + suc k)
  more : forall {n} k -> Compare (n + suc k) n
  same : forall {n} -> Compare n n
```

(a) Define the view function

```
compare : (n m : Nat) -> Compare n m
compare n m = {! !}
```

(b) Now use the view to compute the difference between two numbers

```
difference : Nat -> Nat -> Nat
difference n m = {! !}
```

*Exercise 3.2.* Type checking $\lambda$-calculus

Change the type checker from Section 3.1 to include precise information also in the failing case.

(a) Define inequality on types and change the type comparison to include inequality proofs. Hint: to figure out what the constructors of $\_\neq\_$ should be you can start defining the `_=?=_` function and see what you need from $\_\neq\_$.

```
data _≠_ : Type -> Type -> Set where
  -- ...

data Equal? : Type -> Type -> Set where
  yes : forall {τ} -> Equal? τ τ
  no  : forall {σ τ} -> σ ≠ τ -> Equal? σ τ

_=?=_ : (σ τ : Type) -> Equal? σ τ
σ =?= τ = {! !}
```

(b) Define a type of illtyped terms and change `infer` to return such a term upon failure. Look to the definition of `infer` for clues to the constructors of `BadTerm`.

```
data BadTerm (Γ : Cxt) : Set where
  -- ...

eraseBad : {Γ : Cxt} -> BadTerm Γ -> Raw
eraseBad b = {! !}

data Infer (Γ : Cxt) : Raw -> Set where
  ok  : (τ : Type)(t : Term Γ τ) -> Infer Γ (erase t)
  bad : (b : BadTerm Γ) -> Infer Γ (eraseBad b)

infer : (Γ : Cxt)(e : Raw) -> Infer Γ e
infer Γ e = {! !}
```

*Exercise 3.3.* Properties of list functions

Remember the following predicates on lists from Section 3.1

```
data _∈_ {A : Set}(x : A) : List A -> Set where
  hd : forall {xs} -> x ∈ x :: xs
  tl : forall {y xs} -> x ∈ xs -> x ∈ y :: xs

infixr 30 _::_
data All {A : Set}(P : A -> Set) : List A -> Set where
  []   : All P []
  _::_ : forall {x xs} -> P x -> All P xs -> All P (x :: xs)
```

(a) Prove the following lemma stating that `All` is sound.

```
lemma-All-∈ : forall {A x xs}{P : A -> Set} ->
              All P xs -> x ∈ xs -> P x
lemma-All-∈ p i = {! !}
```

We proved that filter computes a sublist of its input. Now let's finish the job.

(b) Below is the proof that all elements of `filter p xs` satisfies `p`. Doing this without any auxiliary lemmas involves some rather subtle use of with-abstraction.

Figure out what is going on by replaying the construction of the program and looking at the goal and context in each step.

```
lem-filter-sound : {A : Set}(p : A -> Bool)(xs : List A) ->
                   All (satisfies p) (filter p xs)
lem-filter-sound p [] = []
lem-filter-sound p (x :: xs) with inspect (p x)
lem-filter-sound p (x :: xs) | it y prf with p x | prf
lem-filter-sound p (x :: xs) | it .true prf  | true  | refl =
  trueIsTrue prf :: lem-filter-sound p xs
lem-filter-sound p (x :: xs) | it .false prf | false | refl =
  lem-filter-sound p xs
```

(c) Finally prove `filter` complete, by proving that all elements of the original list satisfying the predicate are present in the result.

```
lem-filter-complete : {A : Set}(p : A -> Bool)(x : A){xs : List A} ->
                      x ∈ xs -> satisfies p x -> x ∈ filter p xs
lem-filter-complete p x el px = {! !}
```

*Exercise 3.4.* An XML universe

Here is simplified universe of XML schemas:

```
Tag = String

mutual
  data Schema : Set where
    tag : Tag -> List Child -> Schema

  data Child : Set where
    text : Child
    elem : Nat -> Nat -> Schema -> Child
```

The number arguments to `elem` specifies the minimum and maximum number of repetitions of the subschema. For instance, `elem 0 1 s` would be an optional child and `elem 1 1 s` would be a child which has to be present.

To define the decoding function we need a type of lists of between $n$ and $m$ elements. This is the `FList` type below.

```
data BList (A : Set) : Nat -> Set where
  []   : forall {n} -> BList A n
  _::_ : forall {n} -> A -> BList A n -> BList A (suc n)

data Cons (A B : Set) : Set where
  _::_ : A -> B -> Cons A B

FList : Set -> Nat -> Nat -> Set
FList A zero    m       = BList A m
FList A (suc n) zero    = False
FList A (suc n) (suc m) = Cons A (FList A n m)
```

Now we define the decoding function as a datatype `XML`.

```
mutual
  data XML : Schema -> Set where
    element : forall {kids}(t : Tag) -> All Element kids ->
              XML (tag t kids)

  Element : Child -> Set
  Element text         = String
  Element (elem n m s) = FList (XML s) n m
```

(a) Implement a function to print XML documents. The string concatenation function is `_+++_`.

```
mutual
  printXML : {s : Schema} -> XML s -> String
  printXML xml = {! !}

  printChildren : {kids : List Child} -> All Element kids -> String
  printChildren xs = {! !}
```

## 4 Compiling Agda programs

This section deals with the topic of getting Agda programs to interact with the real world. Type checking Agda programs requires evaluating arbitrary terms, ans as long as all terms are pure and normalizing this is not a problem, but what happens when we introduce side effects? Clearly, we don't want side effects to happen at compile time. Another question is what primitives the language should provide for constructing side effecting programs. In Agda, these problems are solved by allowing arbitrary Haskell functions to be imported as axioms. At compile time, these imported functions have no reduction behaviour, only at run time is the Haskell function executed.

### 4.1 Relating Agda types to Haskell types

In order to be able to apply arbitrary Haskell functions to Agda terms we need to ensure that the run time representation of the Agda terms is the same as what the function expects. To do this we have to tell the Agda compiler about the relationships between our user defined Agda types and the Haskell types used by the imported functions. For instance, to instruct the compiler that our `Unit` type should be compiled to the Haskell unit type `()` we say

```
data Unit : Set where
  unit : Unit

{-# COMPILED_DATA Unit () () #-}
```

The `COMPILED_DATA` directive takes the name of an Agda datatype, the name of the corresponding Haskell datatype and its constructors. The compiler will check that the given Haskell datatype has precisely the given constructors and that their types match the types of the corresponding Agda constructors. Here is the declaration for the maybe datatype:

```
data Maybe (A : Set) : Set where
  nothing : Maybe A
  just : A -> Maybe A

{-# COMPILED_DATA Maybe Maybe Nothing Just #-}
```

Some types have no Agda representation, simply because they are abstract Haskell types exported by some library that we want to use. An example of this is the `IO` monad. In this case we simply postulate the existence of the type and use the `COMPILED_TYPE` directive to tell the compiler how it should be interpreted.

```
postulate IO : Set -> Set
{-# COMPILED_TYPE IO IO #-}
```

The first argument to `COMPILED_TYPE` is the name of the Agda type and the second is the corresponding Haskell type.

### 4.2 Importing Haskell functions

Once the compiler knows what the Agda type corresponding to a Haskell type is, we can import Haskell functions of that type. For instance, we can import the `putStrLn` function to print a string[10] to the terminal.

```
open import Data.String

postulate
  putStrLn : String -> IO Unit

{-# COMPILED putStrLn putStrLn #-}
```

Just as for compiled types the first argument to `COMPILED` is the name of the Agda function and the second argument is the Haskell code it should compile to. The compiler checks that the given code has the Haskell type corresponding to the type of the Agda function.

### 4.3 Our first program

This is all we need to write our first complete Agda program. Here is the `main` function:

```
main : IO Unit
main = putStrLn "Hello world!"
```

To compile the program simply call the command-line tool with the `--compile` (or `-c`) flag. The compiler will compile your Agda program and any Agda modules it imports to Haskell modules and call the Haskell compiler to generate an executable binary.

### 4.4 Haskell module imports

In the example above, everything we imported was defined in the Haskell prelude so there was no need to import any additional Haskell libraries.

[10] The string library contains the compiler directives for how to compile the string type.

This will not be the case in general – for instance, you might write some Haskell code yourself, defining Haskell equivalents of some of your Agda datatypes. To import a Haskell module there is an `IMPORT` directive, which has the same syntax as a Haskell import statement. For instance, to import a function to print a string to standard error, we can write the following:

```
{-# IMPORT System.IO  #-}

postulate printError : String -> IO Unit
{-# COMPILED printError (hPutStrLn stderr) #-}
```

### 4.5 Importing polymorphic functions

As we saw in Section 2.2 in Agda polymorphic functions are modeled by functions taking types as arguments. In Haskell, on the other hand, the type arguments of a polymorphic functions are completely implicit. When importing a polymorphic Haskell function we have to keep this difference in mind. For instance, to import *return* and *bind* of the IO monad we say

```
postulate
  return : {A : Set} -> A -> IO A
  _>>=_  : {A B : Set} -> IO A -> (A -> IO B) -> IO B

{-# COMPILED return (\a -> return) #-}
{-# COMPILED _>>=_  (\a b -> (>>=)) #-}
```

Applications of the Agda functions `return` and `_>>=_` will include the type arguments, so the generated Haskell code must take this into account. Since the type arguments are only there for the benefit of the type checker, the generated code simply throws them away.

### 4.6 Exercises

*Exercise 4.1.* Turn the type checker for $\lambda$-calculus from Section 3.1 into a complete program that can read a file containing a raw $\lambda$-term and print its type if it's well-typed and an error message otherwise. To simplify things you can write the parser in Haskell and import it into the Agda program.

## 5 Further reading

More information on the Agda language and how to obtain the code for these notes can be found on the Agda wiki [2]. If you have any Agda related questions feel free to ask on the Agda mailing list [1].

My thesis [6] contains more of the technical and theoretical details behind Agda, as well as some programming examples. To learn more about dependently typed programming in Agda you can read *The power of Pi* by Oury and Swierstra [8]. For dependently typed programming in general try *The view from the left* by McBride and McKinna [5] and *Why dependent types matter* by Altenkirch, McBride and McKinna [3].

## References


1. The Agda mailing list, 2012. `https://lists.chalmers.se/mailman/listinfo/agda`.
2. The Agda wiki, 2012. `http://www.cs.chalmers.se/~ulfn/Agda`.
3. T. Altenkirch, C. McBride, and J. McKinna. Why dependent types matter. Manuscript, available online, April 2005.
4. P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis·Napoli, 1984.
5. C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, January 2004.
6. U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.
7. U. Norell and J. Chapman. Dependently Typed Programming in Agda (source code), 2012. `http://www.cse.chalmers.se/~ulfn/darcs/AFP08/LectureNotes/`.
8. N. Oury and W. Swierstra. The power of pi. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 39–50, New York, NY, USA, 2008. ACM.